内容

オブジェクト指向―実装

はじめに

Java と“オブジェクト”に関して、Java 言語仕様(SE16)-序章に「汎用、平行[プログラミング]、クラスベース オブジェクト指向言語」であると書かれています。また、章 4.3.1 には「オブジェクトは、あるクラスのインスタンスまたは配列」と定義されています。それがナンなの？と思うかもしれませんが、保守性と品質を確保するためには必要な前提知識であり、それと知って使えばまぁまぁ役に立ちます。

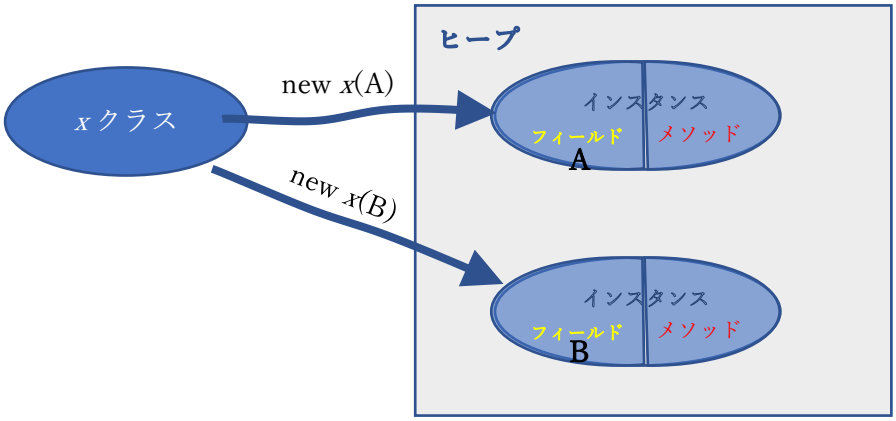
1. インスタンスとクラス 再確認

オブジェクト指向はインスタンスの挙動を中心に考えますが、実装の対象はクラスです。クラスとインスタンスの関係について整理します。

（1）インスタンス

“インスタンス”は「クラスをヒープにコピー[1]して固有の値を持ったもの」です。

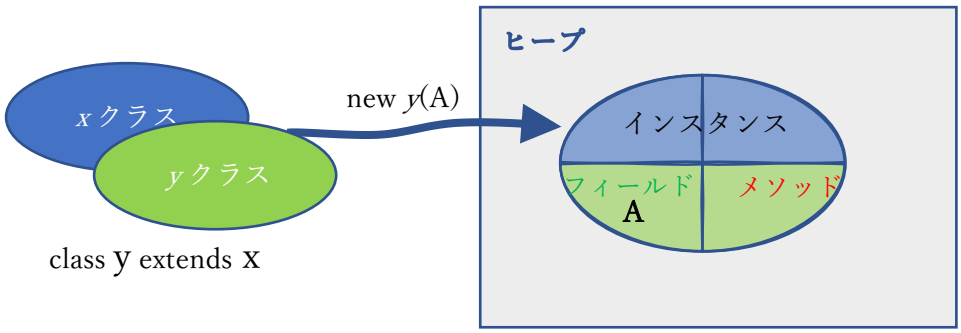
ヒープにはフィールド（初期値を含む）とメソッド（へのポインタ）がコピーされ、コンストラクタやメソッドの呼び出しによりフィールドの値が設定／更新されます。



[1] static フィールド（クラス変数）、static メソッド（クラスメソッド）はコピーされません。また、インスタンスメソッドのコードもメタ情報としてメモリ中一つだけ展開されます。インスタンスが参照する vtable にメソッドのポインタが記録され、実行時に動的バインディングが行われて呼び出すメソッドが決定されます（オーバーライド）
https://www.oracle.com/webfolder/technetwork/jp/javamagazine/Java-SO17-MethodInvoc.pdf

（2）継承関係から作られるインスタンス

継承される側の上位（スーパー）クラス[図のｘクラス]は、継承する側の下位[サブ]クラス[図のｙクラス]が上に重ねられたようにしてインスタンス化されます。



●フィールドは名前で対象が決定される：

・ｘクラスの同一名のフィールドはアクセス修飾子に関係なく外部から[2]は見えなくなります

・但し、ｘクラス（上位）からはｙクラス（下位）のフィールドはアクセスできま**せん**

※アクセス修飾子により開示範囲がきまり、フィールドはクラス単位で管理されます

●メソッドはシグネチャ（メソッド名、引数）で呼び先が決定される[3]：

・ｘクラスの同一シグネチャのメソッドはアクセス修飾子に関係なく外部からは見えなくなり、ｘクラスにキャストしてもｙクラスのメソッドが呼び出されます

・ｘクラス（上位）からもｙクラス（下位）のメソッドが呼び出され**ます**

※アクセス修飾子が public、protected のメソッドはインスタンス単位で管理されます

[2] 宣言しているクラス自身のメソッドからはフィールドが参照できます。また、外部からはキャストすることで上位クラスのフィールドを参照できます

[3] シグネチャが同一のメソッド（ｘクラスをｙクラスでオーバーライド）は戻り値と throw している例外も互換でないとコンパイエラーになります。また、”super.”で修飾すると下位クラスから参照できますが、キャストしても上位クラスのメソッドを呼び出すことはできません。シグネチャが異なる同一名のメソッドも実装可能で、オーバーロードと呼びます

（3）継承関係とインスタンスの互換性

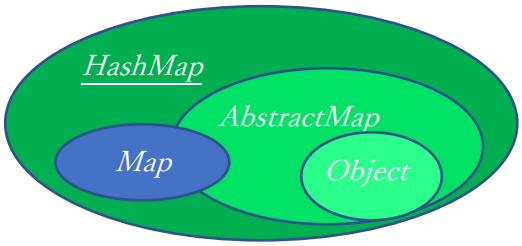
継承関係の下位クラスは、上位クラスと実装宣言したインタフェースの public なフィールドとメソッドを持っていることが保障されるので、上位クラス及び実装(implements)しているインタフェース型[4]として扱うことができます。以下に例を挙げます。

【HashMap】

HashMap は AbstractMap クラスを継承して Map インタフェースを実装しているため、Map として扱うことができます。(同様に AbstractMap、Object としても扱えます)

＜階層ツリー＞

java.lang.Object
　java.util.AbstractMap<K,V>
　　java.util.**HashMap**<K,V>
実装されたインタフェース:
Map<K,V> （その他略）

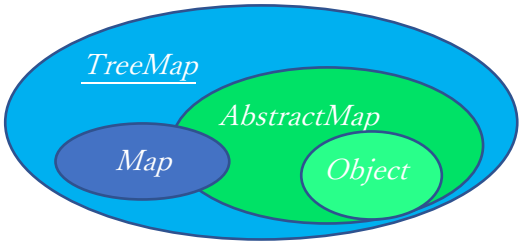


【TreeMap】

TreeMap も同様に AbstractMap クラスを継承して Map インタフェースを実装しています。

＜階層ツリー＞

java.lang.Object
　java.util.AbstractMap<K,V>
　　java.util.**TreeMap**<K,V>
実装されたインタフェース:
Map<K,V> （その他略）



これにより HashMap と TreeMap のインスタンスは両方“Map”として扱うことができます。

［使用例］

```
Map<String, String>　hashM　= new HashMap<String, String> ();
hashM.put("キー", "バリュー");
Map<String, String>　treeM　= new TreeMap<String, String> (hashM);
```

※HashMap、TreeMap の Map のように、外部からはインスタンスの実体がナニか知らなくても使うことができます（多態性の一形態[5]）

[4] Java の”interface”はフィールドとメソッドの宣言だけが書かれたクラスです

[5] オーバーロード、オーバーライドも多態性の実現手段です。

2. 保守性／品質の観点

システムは外部環境の変化に応じた更新が必要なため、容易に改修できる保守性が大事です。

2.1. 一般的な注意点

開発の前提条件として以下の認識が必要です。

① 自分以外の人間が手を入れる

② 改変が可能な製品としての品質が求められる

改修時には以下の不具合が混入・露見しがちです。

① 改変の影響が改修範囲以外に広がる

② 実装の方針によっては改修範囲が広くなりすぎる

2.2. オブジェクトに関連する注意点

オブジェクト／インスタンスの性質からくる注意点もあります。以下に挙げます。

（1）実装の変更が正常に伝播しないケース

＜実行例 1 ＞

JShell で 2 つのクラス Class1 と Class2 を作り、

① Class1.method()は自クラス内で定義した String を返す

② Class2.method()は class1 で定義している Stirng を返す

```
jshell> class Class1 {
   ...>     public static final String ret = "あいうえお";
   ...>     String method(){
   ...>         return ret;
   ...>     }
   ...> }
|  次を作成しました: クラス Class1
jshell> class Class2 {
   ...>     String method(){
   ...>         return Class1.ret;
   ...>     }
   ...> }
|  次を作成しました: クラス Class2
jshell>
jshell> System.out.println((new Class1()).method() + " : " + (new Class2()).method() + " > " + ((new
Class1()).method()==(
new Class2()).method()));
あいうえお : あいうえお > true
```

【①と②の比較結果】

①と②は同一の String に対する参照を返しているので、当然一致します

＜実行例２＞

実行例１の Class1 内で定義している String の内容を変更し、実行例１と同様に実行します。

① Class1.method()は自クラス内で定義した String を返す

② Class2.method()は Class1 で定義している Stirng を返す

```
あいうえお : あいうえお ＞true
＜実行例１の実行直後に、以下＞
jshell> class Class1 {
   ...>     public static final String ret = "かきくけこ";
   ...>     String method(){
   ...>         return ret;
   ...>     }
   ...> }
|  次を変更しました: クラス Class1

jshell> System.out.println((new Class1()).method() + " : " + (new Class2()).method() + " ＞ " + ((new
Class1()).method()==(new Class2()).method()));
かきくけこ : あいうえお ＞false
```

【①と②の比較結果】

①と②は同一の String に対する参照を返しているはずなのに、違う文字列が返されています！

※この例は Class1 で変数を public static final で宣言しているためコンパイラーが参照している側の Class2.method()に変数をコピー（インライン化）し、Class1 側の変更が Class2 に伝播しないためこのようになります（カプセル化の破綻）

（2）private で宣言したフィールドが破壊されるケース

＜実行例＞

Class1 に StringBulder 型の変数 str を private で宣言します（初期値”あいうえお”）。

① Class1::getStr()で str を変数 bef に取得します

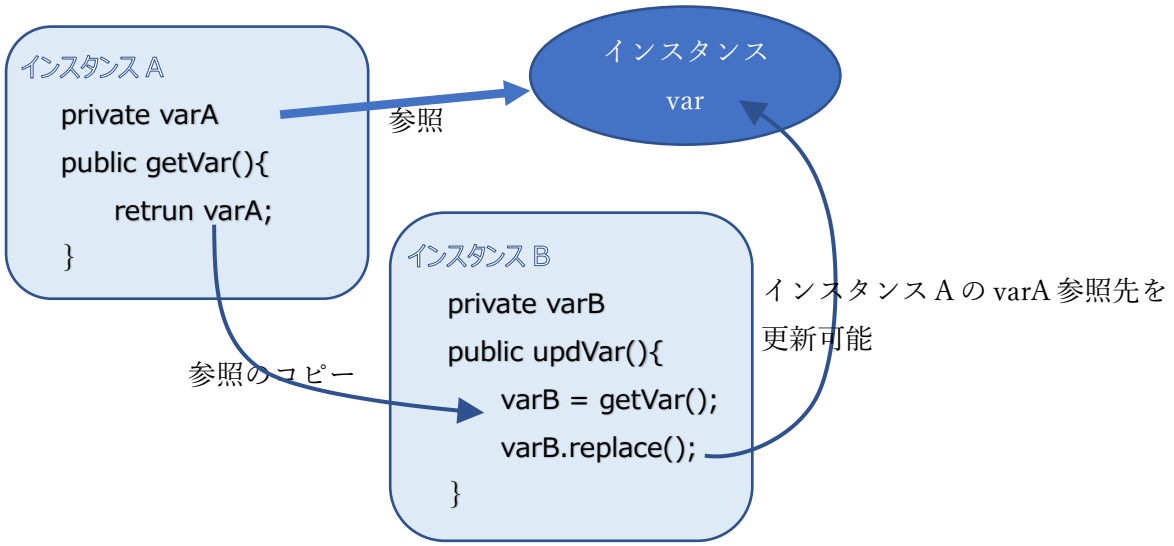
② StringBuilder::replace()で bef の文字列を書き換えます（”あいうえお”⇒“abcdefg”）

③ 再度、Class1::getStr()で str を変数 aft に取得します（“abcdefg”）

⇒ Class1 で private 宣言している内容が書き換わっている

```
jshell> class Class1 {
   ...>     private final StringBuilder str = new StringBuilder("あいうえお");
   ...>     StringBuilder getStr(){
   ...>         return str;
   ...>     }
   ...> }
|  次を作成しました: クラス Class1
jshell>
jshell> Class1 test = new Class1();
test ==> Class1@5419f379
jshell> StringBuilder bef = test.getStr();
bef ==> あいうえお
jshell> bef.replace(0, 5, "abcdefg");
$4 ==> abcdefg
jshell> StringBuilder aft = test.getStr();
aft ==> abcdefg
```

※private で宣言した変数は他のクラスからは見えませんが、変数の中身の“参照”を渡せばその先の参照先インスタンスは更新が可能になり、どこからでも内容の書き換えが可能になります（戻り値だけではなく、引数で渡した参照先も更新可能です）。変数の変更を抑止する final 修飾子を付けても参照先の保護はできません。（カプセル化の破綻）



3. 保守性／品質を上げるには

処理の途中で意図せず変数の値が変わってしまってはデバッグも困難です。保守性を上げるにはインスタンスの強度を上げる必要があり、前項で挙げた問題への対処[6]が必要です。

3.1. コード上の注意

① 不要なフィールドを宣言しない

同一クラスの中のメソッド間でも引数／戻り値で情報の受け渡しをておいた方が後々の改修や再利用が行い易くなります。状態の保存をする必要がないものはローカル変数で宣言してください。

② フィールドと内部メソッドは private で宣言する

private 以外のアクセス修飾子（修飾子無しも含む）は他のクラスから参照可能になり、影響範囲／デバッグの調査範囲も広くなります。外部へ渡す必要がある場合は情報提供用の public メソッド（ゲッター）を用意します。

③ 引数、戻り値はプリミティブ／ラッパークラスか String 型

ｉｎｔ，ｌｏｎｇ等のプリミティブ型やそれらのラッパークラス（Ｉｎｔｅｇｅｒ．Ｌｏｎｇ等）、String は引数、戻り値として受け渡しが行われても元の値が書き換えられることはありません。
また、Collections.unmodifiableCollection()を使えばＭａｐ，Ｌｉｓｔ，Ｓｅｔの変更不可能なインスタンスが取得できます（但し、子階層以降には変更抑止の効力がありません）。

3.2. クラス／メソッドの構造

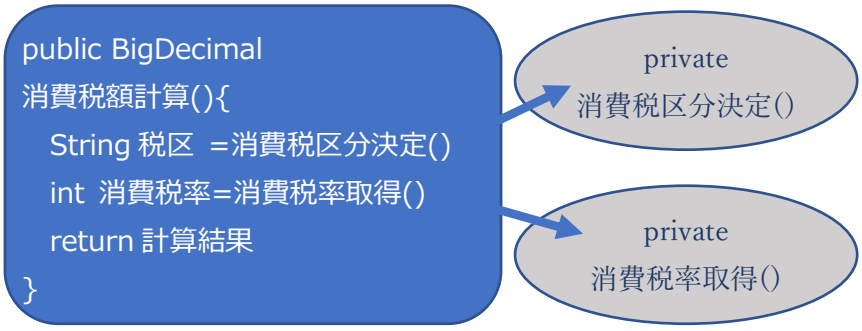
コード上の注意に挙げた項目のいくつかは、メソッドの構成／設計を適切に行うことで意識せずに回避することができます。

① メソッドの階層と処理データの構造を合わせる

適切であれば殆どの変数はローカル変数で事足りるでしょう。

② 一塊の処理を閉じたメソッド群で完結させる

意味のある処理の単位（消費税率の取得ではなく、消費税額計算のように）を 1 public メソッドとそこから呼ばれるいくつかの private メソッドに局所化すれば、中間結果での受け渡しを外部と行う必要がなく処理中のインスタンス変数（が参照している先）の更新を防げます。



[6] 対処には「諦める」という選択もあります。その場合は問題点の全てをテスト観点に入れます。

4. 具体的な実装方法

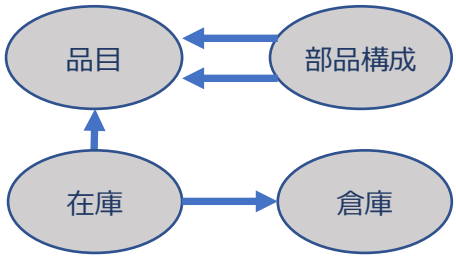
将来の改修（開発中の仕様変更も含む）に伴う品質は、影響箇所を局所化できれば上がります[7]。影響の大きさは①エンティティの関連の変化＞②エンティティの項目追加削除＞③ロジックの変更の順になるので、データの関連[8]に合わせて実装しておけば影響範囲の把握が楽になります。

但し、設計工程で決めた実装方式やフレームワーク（ある場合は）の枠の中での実装になります。

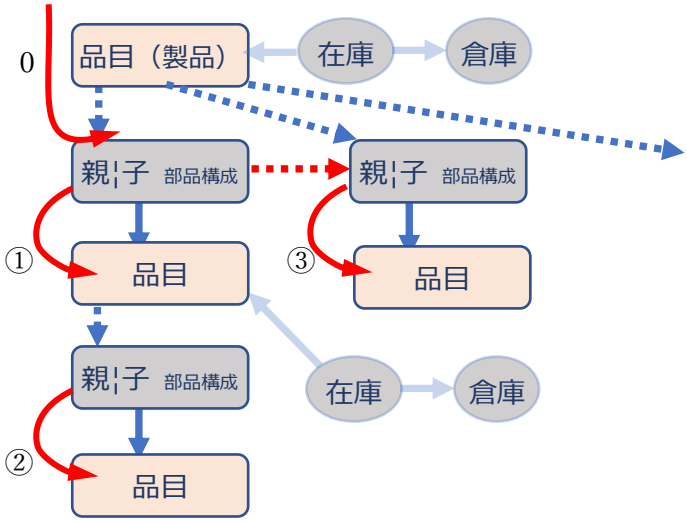
4.1. データ関連とメソッド構成

クラスの中の処理構造は処理対象のデータの関連に合わせます。例として、製品の部品所要量を求める処理を以下に示します。

【エンティティ】



【データ関連】



【要件】

① 受注した製品を部品に展開して、製品の注文数から調達が必要な部品の種類と数を集計する
② 在庫が残っている品目は在庫（倉庫毎に管理）を引き当てて調達必要数から差し引く
③ 製品と部品の諸元は“品目”で管理しており、親品目１単位を作るのに必要な子品目の数（所要量）は“部品構成”で管理している
④ 部品は複数種類の親品目に使われることがある（部品展開中に複数回出現することがある）

---

[7]テストケース数は分岐一つ追加で最低２ケース、二つ追加で２×２の４ケース必要になります。複数のメソッドを改修した場合は更に複雑なテストケース設定が増えるはずです

[8] ここでいう「データの関連」とは、定義としての「エンティティの関連」のことではありません

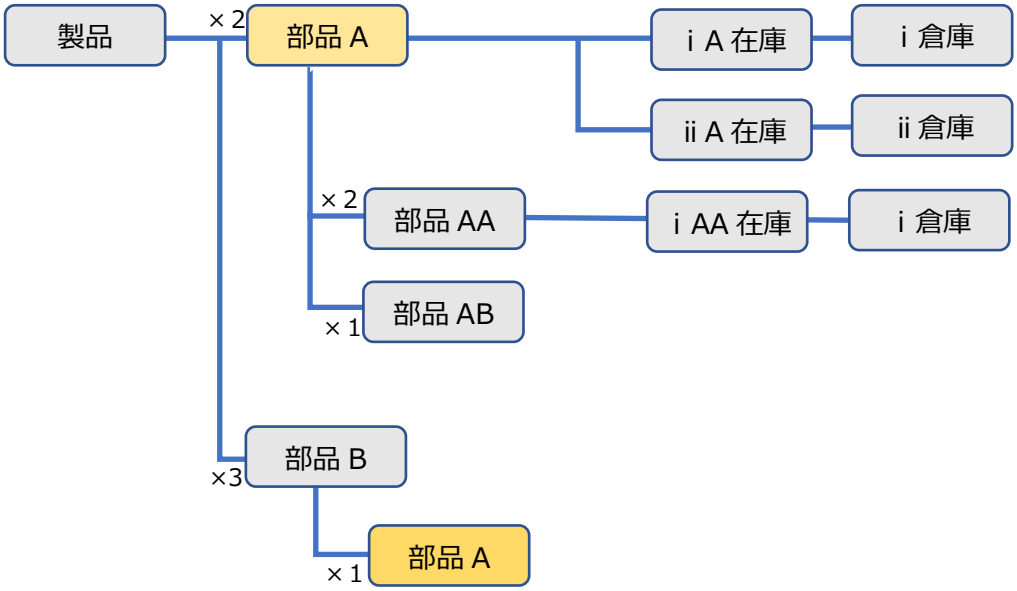
（1）処理の構成方法

データ関連は親品目と子品目を部品構成で関係づける繰り返しになっているため「品目が親部品として登録されている部品構成を読み、所要量を取得する」を繰り返せばよいと分かります。

また、同一品目が複数回出現する可能性があるので在庫の引き当ては部品構成の集計後になります。

【オブジェクト図】※「部品構成」の情報は×ｎで表しています

- 製品は部品Ａ×２、部品Ｂ×３で構成され、部品Ｂは部品Ａを１つ使っている
- 部品Ａは部品ＡＡ×２、部品ＡＢ×１で構成されていて、ｉ倉庫とii倉庫に在庫がある



① クラス／メソッドの初期設計

エンティティとメソッドの関係が１：１かｎ：１となるようにし、１つのエンティティを複数のメソッドから参照しないように実装します。また、パラメタや戻り値はローカル変数で宣言します。

public 所要量集計 メソッド

① 部品構成メソッド を実行

private 部品構成 メソッド

a. var 子品目毎所要量 =0 //戻り値宣言＆初期化

b. パラメタの品目が親部品になっている部品構成を全件読み込む

c. 繰り返し：部品構成

部品構成 1 件処理

・部品構成の所要量を子品目毎所要量に足しこみ

・子品目を引数に「部品構成処理」を呼び出し［再帰］

・戻り値の子品目毎所要量をローカルの子品目毎所要量にマージ

d. 子品目毎所要量を戻り値に return

② 繰り返し：戻り値の子品目毎所要量の品目

private 品目 メソッド

a. パラメタの品目を読込み、品目名等を取得

b. 在庫集計処理

private 在庫 メソッド

・var 在庫数 = 0 //戻り値宣言＆初期化

・パラメタの品目に対する在庫を全件読み込む

・集計：在庫数

・在庫数を戻り値に return

c. 品目名、在庫数を戻り値(Map)にして return

③ 品目コード、品目名、所要量-在庫数(CSV 形式)を戻り値に return

注意：部品の階層や 1 製品当たりの部品品目数が多いと再帰処理での実装は危険です。階層が深いとスタックオーバフローが発生し、部品数が多いとヒープの枯渇で OutOfMemory の例外が発生します。中間表を作るか、ストアドプロシージャで実装する方が良いかもしれません

② クラス／メソッドの設計変更

作成したクラス／メソッドの構成が適正か判断するには、発生し得る仕様の変更等に対応できるかシミュレートしてみることが大事です。

【変更要件】製品構成中に同一部品が複数回現れた場合、全く同じ結果になるにもかかわらず毎回末端部品まで処理が発生する。主要部品は子部品数が多い上に出現頻度が高く時間が掛かりすぎるため、品目エンティティに「子品目所要量展開要」のフラグを持つこととする。

【対応案】部品構成メソッドの中で品目メソッドを実行し、子品目について品目メソッドが未実行かつ「子品目所要量展開要」のとき在庫集計を行うようにする。
また、所要量集計メソッドで品目情報（品目コード、品目名、所要量、在庫数）を管理し、部品構成メソッドの戻り値に品目名、在庫数を追加する。

public 所要量集計 メソッド

0 var 品目情報（品目コード,品目名、所要量、在庫数）初期化

① 部品構成メソッド を実行（パラメタに品目情報）

private 部品構成 メソッド

a. var 子品目毎所要量 =0 //戻り値宣言＆初期化

b. パラメタの品目が親部品になっている部品構成を全件読み込む

c. 繰り返し：部品構成

部品構成 1 件処理

・品目メソッドを実行する(品目情報)

・部品構成の所要量を子品目毎所要量に足しこみ

・子品目を引数に「部品構成処理」を呼び出し［再帰］

・戻り値の子品目毎所要量をローカルの子品目毎所要量にマージ

d.子品目毎所要量を戻り値に return

~~②・繰り返し：戻り値の子品目毎所要量の品目~~

private 品目 メソッド（品目情報）

a. パラメタの品目を読込み、品目名等を取得

b. 「子品目所要量展開要」かつ品目情報=null のとき在庫集計処理

private 在庫 メソッド

・var 在庫数 = 0 //戻り値宣言＆初期化

・パラメタの品目に対する在庫を全件読み込む

・集計：在庫数

・在庫数を戻り値に return

c. 品目名、在庫数を戻り値(Map)にして return

③ 品目コード、品目名、所要量-在庫数(CSV 形式)を戻り値に return

【変更案　評価】

設計変更の対応案シミュレーション結果は、実現可能にみえる。

ただし、このままの実装だと 品目情報を「所要量計算」⇒「部品構成」⇒「品目」と 2 階層持ちまわることになり、結合度が強くなっている。もう少し検討が必要かも…

（2）デザインパターン／フレームワークと名前の重要性

よく使われる有用な実装のいくつかが類型化されパターン名が付けられています（GoF と呼ばれる人たちの書いた本が有名で、前項の例は Composite パターンと呼ばれたものが似ています）。

多くのパターンの中には「だれだって普通そう作るだろ」といったものもありますが、重要なのは名前が付いているということです。「ｘｘパターン」というだけで協働者が共通認識を持てる（＝定義されている）ことです。クラス／フィールド／メソッドの名前を「わかる」ように付けるのは非常に重要です。

また、フレームワークを使った実装の場合はクラス名やメソッド名が外部の環境定義と関係づけられているものがあります。この場合、誤った名前だと正常に動作しません。

4.2. データ構造によらない実装

（1）ログやエラー処理

業務的なロジックとは別に、ログやエラー処理の実装内容を一般的に方式として決定し共通化します。これらの共通処理はユーティリティクラスとしたり基底クラスとして実装します。

```
class 方式共通処理 {
    protected void エラー処理() {
    }
    protected void ログ出力() {
    }
}
```

```
class 業務処理 extends 共通処理 {
        ：
    エラー処理(errCode, “エラーメッセージ");
    ログ出力((errCode, “エラーメッセージ");
}
```

（2）システムの状態や時刻・カレンダー

棚卸中で在庫の情報が使えない／当日の注文受付締め切り時間を過ぎている等、環境や状態によりサービス内容を変更しなければいけない場合があります。また、長時間かかったり大量のデーをロックしてしまうような処理ではシステムの状態変化やキャンセルの受付を途中で感知できる構成が必要になるかもしれません。

4.3. オブジェクトの特性（多態性）を使った実装

多態性を利用したコードはデザインパターンとして数多く公開されています。下の例は取引先の種類と品目により異なった率を適用して割引額を計算しようとするものです。

●取引先クラス（デフォルト＋２種類）

```
class 取引先 {
    private String whoami = "取引先";
    public int get 優遇割引率(){
        return 0;
    }
    public String toString() {
        return whoami;
    }
}
class 一般客 extends 取引先 {
    private String whoami = "一般客";
    public String toString() {
        return whoami;
    }
}
class 優遇客 extends 取引先 {
    private String whoami = "優遇客";
    public int get 優遇割引率(){
        return 10;
    }
    public String toString() {
        return whoami;
    }
}
```

継承

●品目クラス（デフォルト＋２種類）

```
class 品目 {
    private String whoami = "品目";
    public String toString() {
        return whoami;
    }
}
class 通常品 extends 品目 {
    private String whoami = "通常品";
    public String toString() {
        return whoami;
    }
}
class 特選品 extends 品目 {
    private String whoami = "特選品";
    public String toString() {
        return whoami;
    }
}
```

●手数料クラス（取引先×品目、一般客×通常品、優遇客×特選品）

```
class 手数料 {
    public static int 計算(取引先 取引先, 品目 品目, int 価格){
        System.out.println("式 1 取引先=" + 取引先.toString() + " | 品目=" + 品目.toString() + " | 割引率=" + 取引先.get 優遇割引率());
        return (価格 * (取引先.get 優遇割引率() + 3) /100);
    }
    public static int 計算(一般客 取引先, 通常品 品目, int 価格){
        System.out.println("式 2 取引先=" + 取引先.toString() + " | 品目=" + 品目.toString() + " | 割引率=" + 取引先.get 優遇割引率());
        return (価格 * (取引先.get 優遇割引率() + 1) /100);
    }
    public static int 計算(優遇客 取引先, 特選品 品目, int 価格){
        System.out.println("式 3 取引先=" + 取引先.toString() + " | 品目=" + 品目.toString() + " | 割引率=" + 取引先.get 優遇割引率());
        return (価格 * (取引先.get 優遇割引率() + 10) /100);
    }
}
```


● 実行

手数料クラスに定義されているメソッドと、実行呼び出しの組み合わせは以下のとおりです。

実装メソッド：①取引先×品目、②一般客×通常品、③優遇客×特選品

実行呼び出し：A. 一般客×通常品、B. 一般客×特選品、C. 優遇客×通常品、D.優遇客×特選品

実行メソッド：　A:②　B:①　C:①　D:③

👉Java が条件に合うメソッドを選択[9]します。

※JShell で実行した結果は以下のとおりです。

```
jshell> 一般客 一般客 = new 一般客();
一般客 ==> 一般客

jshell> 優遇客 優遇客 = new 優遇客();
優遇客 ==> 優遇客

jshell> 通常品 通常品 = new 通常品();
通常品 ==> 通常品

jshell> 特選品 特選品 = new 特選品();
特選品 ==> 特選品

jshell> 手数料.計算(一般客, 通常品, 10000);
式 2 取引先=一般客 | 品目=通常品 | 割引率=0
$47 ==> 100

jshell> 手数料.計算(一般客, 特選品, 10000);
式 1 取引先=一般客 | 品目=特選品 | 割引率=0
$48 ==> 500

jshell> 手数料.計算(優遇客, 通常品, 10000);
式 1 取引先=優遇客 | 品目=通常品 | 割引率=10
$49 ==> 1500

jshell> 手数料.計算(優遇客, 特選品, 10000);
式 3 取引先=優遇客 | 品目=特選品 | 割引率=10
$50 ==> 2000
```

※注意　例示したソースではクラス名とメソッド名に日本語を使っていますが、Windows が混在する開発環境で日本語を使用すると文字化けの原因になります（各プロジェクトのコーディング規約に従ってください）。

[9] シグネチャが完全に一致するメソッドが見つからない場合、“Most Specific Method”（最も具体的なメソッド）が選ばれる https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html

（1）効果

取引先と品目の 2 要素で値が決まる場合「取引先の種類数 × 品目の種類数」の組み合わせが発生し、コード化すると組み合わせ分の分岐が必要になります。一方、このように多態性を使うと条件分岐がなくなりテスト／保守の負荷が軽減する可能性があります。

（2）その他のポイント（汎化）

この例のように継承関係やインタフェースの実装があると、具体的なクラス（「一般客」や「優遇客」）を汎化したクラス（「取引先」）として取り扱うことができます。これは引数や戻り値を集約して使うことができ、取引先にや品目の種類（例えば「紹介客」、「キャンペーン品」等）が増えてもクラス間のインタフェース（コーリングシーケンス）に影響がでません。

（3）デザインパターン／フレームワークとの関係

本項の例は Double Dispatch というパターンの一部を切り出したものです。デザインパターンを使うと品質の向上に期待できる一方で硬直的になり仕様変更が難しくなる可能性もあります。本例では割引条件の追加や特殊な取り扱いが必要な分類（顧客集団等）が発生した場合、既存のクラスを含めた全面的な改修が必要になります。

業務処理の場合は開発時点での品質は上がるかもしれませんが、将来の保守性が落ちてしまう可能性もあるので組織の将来動向を見越した判断が必要です。また、フレームワークを使った開発の場合はパターンの一部として実装することもあります。

業務色が無い、制御処理等の場合はデザインパターンの利用は有効で Singleton のようにパターンを使わないと実現できない機能もあります。

5. 実装時に追加のクラスや定義ファイル

Java で匿名クラスや内部クラスを使うとコンパイル時に”*クラス名*＋$n.class”という名前のファイルが自動的に作られます。それ以外にもデザインパターン等を適用した実装ではインタフェースが多用されるため、作成者でさえ認識していないファイルが作られている可能性があります。

また、設計工程でのクラス分割とは別に実装時のクラス分割（Java ソース、class ファイル）や、実行のためのプロパティ・ファイル、環境定義ファイルの作成が必要になる場合があります。

統合テスト環境への反映ではいずれも必要になるファイルなので、以下の手順が必要です。

① 単体テストが終わる都度、漏れなくリポジトリに最新資材を反映する
② 運用時に必要になる環境定義については、設計ドキュメント等に定義内容を反映する

6. リファクタリング

実装行程では上流工程での変更（設計、要件、実現手法）が全て影響するため、全面的な作り直しが 2～3 度発生するのはよくあることです。また、実装工程自体でもしばしば試行錯誤的な改修を行う場合があります。

機能要件が満たせる段階になったら、以下の観点でソースを全体的に見直す必要があります。

① コメントに誤りがないか（着手時と完成時で意味が変わってないか）
② 変数名（フィールド、ローカル変数）、メソッド名は定義目的と合っているか（変わってないか）
③ 使わなくなったり、到達不能なロジックは残っていないか
④ 非効率なロジックは残っていないか（実装時期による実現方法の違いはないか）
⑤ 同一のことを行っているロジックが複数ないか（一つの事実は一つの場所に記述する）

7. 実装着手前に必要な事項

新システムの開発では、基盤のミドルウェアやフレームワークになにを、どの範囲で使うか（以下、“方式”）を決める必要があります。方式は機能性（必要な機能を持っているか）や将来性（開発中にサポート停止になったりしないか）が重要ですが、生産性にも大きく影響します。実装への着手には以下の事項について確認が必要です。

① ログ出力、エラー処理の手順、ユーティリティクラスや基底クラスと個別実装の役割分担
＜特にフレームワークを使う場合は、具体的な制約や実装方針の周知・確認＞
② 永続化データを使用する場合は、API（バウンダリー）の使用方法
③ 割り込みや再実行、並行処理に関して実装上の考慮が必要であればその内容
④ 実装工程で作成した資材（特に以下のもの）の運用ルール
・ソース
・コンパイル生成物
・実行時に必要な環境定義ファイル等
・設計書へのフィードバック

以上

